C interfaces to GALAHAD LLSR

Jari Fowkes and Nick Gould
STFC Rutherford Appleton Laboratory

Thu Jun 22 2023

Chapter 1

GALAHAD C package llsr

1.1 Introduction

1.1.1 Purpose

Given a real m by n matrix A, a real n by n symmetric diagonally dominant-matrix S, a real m vector b and scalars $\sigma > 0$ and $p >= 2$, this package finds a **minimizer of the regularized linear least-squares objective function**

$$1/2\|Ax - b\|_2^w + \sigma/p\|x\|_S^p,$$

where the S-norm of x is $\|x\|_S = \sqrt{x^T Sx}$. This problem commonly occurs as a subproblem in nonlinear least-squares calculations. The matrix S need not be provided in the commonly-occurring ℓ_2-regularization case for which $S = I$, the n by n identity matrix.

Factorization of matrices of the form

$$(1) \quad K(\lambda) = \begin{pmatrix} \lambda S & A^T \\ A & -I \end{pmatrix}$$

of scalars λ will be required, so this package is most suited for the case where such a factorization may be found efficiently. If this is not the case, the GALAHAD package `LSRT` may be preferred.

1.1.2 Authors

N. I. M. Gould, STFC-Rutherford Appleton Laboratory, England.

C interface, additionally J. Fowkes, STFC-Rutherford Appleton Laboratory.

1.1.3 Originally released

June 2023, C interface June 2023.

1.1.4 Terminology

The required solution x_* necessarily satisfies the optimality condition $A^T Ax_* + \lambda_* Sx_* = A^T b$, where $\lambda_* = \sigma\|x_*\|^{p-2}$.

1.1.5 Method

The method is iterative, and proceeds in two phases. Firstly, lower and upper bounds, λ_L and λ_U, on λ_* are computed using Gershgorin's theorems and other eigenvalue bounds, including those that may involve the Cholesky factorization of S. The first phase of the computation proceeds by progressively shrinking the bound interval $[\lambda_L, \lambda_U]$ until a value λ for which $\|x(\lambda)\|_S \geq \sigma\|x(\lambda)\|_S^{p-2}$ is found. Here $x(\lambda)$ and its companion $y(\lambda)$ are defined to be a solution of Once the terminating λ from the first phase has been discovered, the second phase consists of applying Newton or higher-order iterations to the nonlinear \`\`secular'' equation $\lambda = \sigma\|x(\lambda)\|_S^{p-2}$ with the knowledge that such iterations are both globally and ultimately rapidly convergent.

The dominant cost is the requirement that we solve a sequence of linear systems (2). This may be rewritten as

$$(3) \quad \begin{pmatrix} \lambda S & A^T \\ A & -I \end{pmatrix} \begin{pmatrix} x(\lambda) \\ y(\lambda) \end{pmatrix} = \begin{pmatrix} A^T b \\ 0 \end{pmatrix}$$

for some auxiliary vector $y(\lambda)$. In general a sparse symmetric, indefinite factorization of the coefficient matrix $K(\lambda)$ of (3) is often preferred to a Cholesky factorization of that of (2).

1.1.6 Reference

The method is the obvious adaptation to the linear least-squares problem of that described in detail in

H. S. Dollar, N. I. M. Gould and D. P. Robinson. On solving trust-region and other regularised subproblems in optimization. Mathematical Programming Computation **2(1)** (2010) 21–57.

1.1.7 Call order

To solve a given problem, functions from the llsr package must be called in the following order:

- llsr_initialize - provide default control parameters and set up initial data structures
- llsr_read_specfile (optional) - override control values by reading replacement values from a file
- llsr_import - set up problem data structures and fixed values
- llsr_import_scaling (optional) - set up problem data structures for S if required
- llsr_reset_control (optional) - possibly change control parameters if a sequence of problems are being solved
- llsr_solve_problem - solve the regularization problem
- llsr_information (optional) - recover information about the solution and solution process
- llsr_terminate - deallocate data structures

See Section 4.1 for examples of use.

1.1.8 Unsymmetric matrix storage formats

The unsymmetric m by n constraint matrix A may be presented and stored in a variety of convenient input formats.

Both C-style (0 based) and fortran-style (1-based) indexing is allowed. Choose `control.f_indexing` as `false` for C style and `true` for fortran style; the discussion below presumes C style, but add 1 to indices for the corresponding fortran version.

Wrappers will automatically convert between 0-based (C) and 1-based (fortran) array indexing, so may be used transparently from C. This conversion involves both time and memory overheads that may be avoided by supplying data that is already stored using 1-based indexing.

1.1.8.1 Dense storage format

The matrix A is stored as a compact dense matrix by rows, that is, the values of the entries of each row in turn are stored in order within an appropriate real one-dimensional array. In this case, component $n * i + j$ of the storage array A_val will hold the value A_{ij} for $0 \leq i \leq m-1, 0 \leq j \leq n-1$.

1.1.8.2 Sparse co-ordinate storage format

Only the nonzero entries of the matrices are stored. For the l-th entry, $0 \leq l \leq ne-1$, of A, its row index i, column index j and value A_{ij}, $0 \leq i \leq m-1$, $0 \leq j \leq n-1$, are stored as the l-th components of the integer arrays A_row and A_col and real array A_val, respectively, while the number of nonzeros is recorded as A_ne = ne.

1.1.8.3 Sparse row-wise storage format

Again only the nonzero entries are stored, but this time they are ordered so that those in row i appear directly before those in row i+1. For the i-th row of A the i-th component of the integer array A_ptr holds the position of the first entry in this row, while A_ptr(m) holds the total number of entries. The column indices j, $0 \leq j \leq n-1$, and values A_{ij} of the nonzero entries in the i-th row are stored in components l = A_ptr(i), . . ., A_ptr(i+1)-1, $0 \leq i \leq m-1$, of the integer array A_col, and real array A_val, respectively. For sparse matrices, this scheme almost always requires less storage than its predecessor.

1.1.9 Symmetric matrix storage formats

Likewise, the non-trivial symmetric n by n scaling matrix S may be presented and stored in a variety of formats. But crucially symmetry is exploited by only storing values from the lower triangular part (i.e, those entries that lie on or below the leading diagonal).

1.1.9.1 Dense storage format

The matrix S is stored as a compact dense matrix by rows, that is, the values of the entries of each row in turn are stored in order within an appropriate real one-dimensional array. Since S is symmetric, only the lower triangular part (that is the part s_{ij} for $0 \leq j \leq i \leq n-1$) need be held. In this case the lower triangle should be stored by rows, that is component $i * i/2 + j$ of the storage array S_val will hold the value s_{ij} (and, by symmetry, s_{ji}) for $0 \leq j \leq i \leq n-1$.

1.1.9.2 Sparse co-ordinate storage format

Only the nonzero entries of the matrices are stored. For the l-th entry, $0 \leq l \leq ne-1$, of S, its row index i, column index j and value s_{ij}, $0 \leq j \leq i \leq n-1$, are stored as the l-th components of the integer arrays S_row and S_col and real array S_val, respectively, while the number of nonzeros is recorded as S_ne = ne. Note that only the entries in the lower triangle should be stored.

1.1.9.3 Sparse row-wise storage format

Again only the nonzero entries are stored, but this time they are ordered so that those in row i appear directly before those in row i+1. For the i-th row of S the i-th component of the integer array S_ptr holds the position of the first entry in this row, while S_ptr(n) holds the total number of entries. The column indices j, $0 \leq j \leq i$, and values s_{ij} of the entries in the i-th row are stored in components l = S_ptr(i), . . ., S_ptr(i+1)-1 of the integer array S_col, and real array S_val, respectively. Note that as before only the entries in the lower triangle should be stored. For sparse matrices, this scheme almost always requires less storage than its predecessor.

1.1.9.4 Diagonal storage format

If S is diagonal (i.e., $s_{ij} = 0$ for all $0 \leq i \neq j \leq n-1$) only the diagonals entries s_{ii}, $0 \leq i \leq n-1$ need be stored, and the first n components of the array S_val may be used for the purpose.

Chapter 2

File Index

2.1 File List

Here is a list of all files with brief descriptions:

Chapter 3

File Documentation

3.1 galahad_llsr.h File Reference

```
#include <stdbool.h>
#include "galahad_precision.h"
#include "galahad_cfunctions.h"
#include "galahad_sbls.h"
#include "galahad_sls.h"
#include "galahad_ir.h"
```

Data Structures

- struct llsr_control_type
- struct llsr_time_type
- struct llsr_history_type
- struct llsr_inform_type

Functions

- void llsr_initialize (void **data, struct llsr_control_type *control, int *status)
- void llsr_read_specfile (struct llsr_control_type *control, const char specfile[ ])
- void llsr_import (struct llsr_control_type *control, void **data, int *status, int m, int n, const char A_type[ ], int A_ne, const int A_row[ ], const int A_col[ ], const int A_ptr[ ])
- void llsr_import_scaling (struct llsr_control_type *control, void **data, int *status, int n, const char S_type[ ], int S_ne, const int S_row[ ], const int S_col[ ], const int S_ptr[ ])
- void llsr_reset_control (struct llsr_control_type *control, void **data, int *status)
- void llsr_solve_problem (void **data, int *status, int m, int n, const real_wp_ power, const real_wp_ weight, int A_ne, const real_wp_ A_val[ ], const real_wp_ b[ ], real_wp_ x[ ], int S_ne, const real_wp_ S_val[ ])
- void llsr_information (void **data, struct llsr_inform_type *inform, int *status)
- void llsr_terminate (void **data, struct llsr_control_type *control, struct llsr_inform_type *inform)

3.1.1 Data Structure Documentation

3.1.1.1 struct llsr_control_type

control derived type as a C struct

Examples

llsrt.c, and llsrtf.c.

Data Fields

bool	f_indexing	use C or Fortran sparse matrix indexing
int	error	unit for error messages
int	out	unit for monitor output
int	print_level	controls level of diagnostic output
int	new_a	how much of A has changed since the previous call. Possible values are • 0 unchanged • 1 values but not indices have changed • 2 values and indices have changed
int	new_s	how much of S has changed since the previous call. Possible values are • 0 unchanged • 1 values but not indices have changed • 2 values and indices have changed
int	max_factorizations	the maximum number of factorizations (=iterations) allowed. -ve implies no limit
int	taylor_max_degree	maximum degree of Taylor approximant allowed ($<=$ 3)
real_wp_	initial_multiplier	initial estimate of the Lagrange multipler
real_wp_	lower	lower and upper bounds on the multiplier, if known
real_wp_	upper	see lower
real_wp_	stop_normal	stop when $\|\|x\| - (\lambda/\sigma)^{1/(p-2)}\| \leq$ stop_normal $*$ max(1, $\|x\|$)
bool	use_initial_multiplier	ignore initial_multiplier?
bool	space_critical	if space is critical, ensure allocated arrays are no bigger than needed
bool	deallocate_error_fatal	exit if any deallocation fails
char	definite_linear_solver[31]	definite linear equation solver
char	prefix[31]	all output lines will be prefixed by prefix(2:LEN(TRIM(.prefix))-1) where prefix contains the required string enclosed in quotes, e.g. "string" or 'string'
struct sbls_control_type	sbls_control	control parameters for the symmetric factorization and related linear solves (see sbls_c documentation)
struct sls_control_type	sls_control	control parameters for the factorization of S and related linear solves (see sls_c documentation)
struct ir_control_type	ir_control	control parameters for iterative refinement for definite system solves (see ir_c documentation)

3.1.1.2 struct llsr_time_type

time derived type as a C struct

Data Fields

real_wp_	total	total CPU time spent in the package
real_wp_	assemble	CPU time assembling $K(\lambda)$ in (1)
real_wp_	analyse	CPU time spent analysing $K(\lambda)$.
real_wp_	factorize	CPU time spent factorizing $K(\lambda)$.
real_wp_	solve	CPU time spent solving linear systems inolving $K(\lambda)$.
real_wp_	clock_total	total clock time spent in the package
real_wp_	clock_assemble	clock time assembling $K(\lambda)$
real_wp_	clock_analyse	clock time spent analysing $K(\lambda)$
real_wp_	clock_factorize	clock time spent factorizing $K(\lambda)$
real_wp_	clock_solve	clock time spent solving linear systems inolving $K(\lambda)$

3.1.1.3 struct llsr_history_type

history derived type as a C struct

Data Fields

real_wp_	lambda	the value of λ
real_wp_	x_norm	the corresponding value of $\|x(\lambda)\|_M$
real_wp_	r_norm	the corresponding value of $\|Ax(\lambda) - b\|_2$

3.1.1.4 struct llsr_inform_type

inform derived type as a C struct

Examples

llsrt.c, and llsrtf.c.

Data Fields

int	status	reported return status: • 0 the solution has been found • -1 an array allocation has failed • -2 an array deallocation has failed • -3 n and/or Delta is not positive • -10 the factorization of $K(\lambda)$ failed • -15 S does not appear to be strictly diagonally dominant • -16 ill-conditioning has prevented furthr progress
int	alloc_status	STAT value after allocate failure.

Data Fields

int	factorizations	the number of factorizations performed
int	len_history	the number of ($\|x\|_S$, λ) pairs in the history
real_wp_	r_norm	corresponding value of the two-norm of the residual, $\|Ax(\lambda) - b\|$
real_wp_	x_norm	the S-norm of x, $\|x\|_S$
real_wp_	multiplier	the multiplier corresponding to the regularization term
char	bad_alloc[81]	name of array which provoked an allocate failure
struct llsr_time_type	time	time information
struct llsr_history_type	history[100]	history information
struct sbls_inform_type	sbls_inform	information from the symmetric factorization and related linear solves (see sbls_c documentation)
struct sls_inform_type	sls_inform	information from the factorization of S and related linear solves (see sls_c documentation)
struct ir_inform_type	ir_inform	information from the iterative refinement for definite system solves (see ir_c documentation)

3.1.2 Function Documentation

3.1.2.1 llsr_initialize()

```
void llsr_initialize (
            void ** data,
            struct llsr_control_type * control,
            int * status )
```

Set default control values and initialize private data

Parameters

in,out	*data*	holds private internal data
out	*control*	is a struct containing control information (see llsr_control_type)
out	*status*	is a scalar variable of type int, that gives the exit status from the package. Possible values are (currently): • 0. The import was succesful.

Examples

llsrt.c, and llsrtf.c.

3.1.2.2 llsr_read_specfile()

```
void llsr_read_specfile (
            struct llsr_control_type * control,
            const char specfile[] )
```

Read the content of a specification file, and assign values associated with given keywords to the corresponding control parameters

Parameters

in,out	*control*	is a struct containing control information (see llsr_control_type)
in	*specfile*	is a character string containing the name of the specification file

3.1.2.3 llsr_import()

```
void llsr_import (
            struct llsr_control_type * control,
            void ** data,
            int * status,
            int m,
            int n,
            const char A_type[],
            int A_ne,
            const int A_row[],
            const int A_col[],
            const int A_ptr[] )
```

Import problem data into internal storage prior to solution.

Parameters

in	*control*	is a struct whose members provide control paramters for the remaining prcedures (see llsr_control_type)
in,out	*data*	holds private internal data
in,out	*status*	is a scalar variable of type int, that gives the exit status from the package. Possible values are: • 1. The import was succesful, and the package is ready for the solve phase • -1. An allocation error occurred. A message indicating the offending array is written on unit control.error, and the returned allocation status and a string containing the name of the offending array are held in inform.alloc_status and inform.bad_alloc respectively. • -2. A deallocation error occurred. A message indicating the offending array is written on unit control.error and the returned allocation status and a string containing the name of the offending array are held in inform.alloc_status and inform.bad_alloc respectively. • -3. The restriction n $>$ 0 or requirement that type contains its relevant string 'dense', 'coordinate', 'sparse_by_rows', 'diagonal' or 'absent' has been violated.

Parameters

in	*m*	is a scalar variable of type int, that holds the number of residuals, i.e., the number of rows of A. m must be positive.
in	*n*	is a scalar variable of type int, that holds the number of variables, i.e., the number of columns of A. n must be positive.
in	*A_type*	is a one-dimensional array of type char that specifies the unsymmetric storage scheme used for the constraint Jacobian, A if any. It should be one of 'coordinate', 'sparse_by_rows' or 'dense'; lower or upper case variants are allowed.
in	*A_ne*	is a scalar variable of type int, that holds the number of entries in A, if used, in the sparse co-ordinate storage scheme. It need not be set for any of the other schemes.
in	*A_row*	is a one-dimensional array of size A_ne and type int, that holds the row indices of A in the sparse co-ordinate storage scheme. It need not be set for any of the other schemes, and in this case can be NULL.
in	*A_col*	is a one-dimensional array of size A_ne and type int, that holds the column indices of A in either the sparse co-ordinate, or the sparse row-wise storage scheme. It need not be set when the dense or diagonal storage schemes are used, and in this case can be NULL.
in	*A_ptr*	is a one-dimensional array of size n+1 and type int, that holds the starting position of each row of A, as well as the total number of entries, in the sparse row-wise storage scheme. It need not be set when the other schemes are used, and in this case can be NULL.

Examples

llsrt.c, and llsrtf.c.

3.1.2.4 llsr_import_scaling()

```
void llsr_import_scaling (
            struct llsr_control_type * control,
            void ** data,
            int * status,
            int n,
            const char S_type[ ],
            int S_ne,
            const int S_row[ ],
            const int S_col[ ],
            const int S_ptr[ ] )
```

Import the scaling matrix S into internal storage prior to solution. Thus must have been preceeded by a call to llsr_import.

Parameters

in	*control*	is a struct whose members provide control paramters for the remaining prcedures (see llsr_control_type)
in,out	*data*	holds private internal data

Parameters

in,out	*status*	is a scalar variable of type int, that gives the exit status from the package. Possible values are: • 1. The import was succesful, and the package is ready for the solve phase • -1. An allocation error occurred. A message indicating the offending array is written on unit control.error, and the returned allocation status and a string containing the name of the offending array are held in inform.alloc_status and inform.bad_alloc respectively. • -2. A deallocation error occurred. A message indicating the offending array is written on unit control.error and the returned allocation status and a string containing the name of the offending array are held in inform.alloc_status and inform.bad_alloc respectively. • -3. The restriction n $>$ 0 or requirement that type contains its relevant string 'dense', 'coordinate', 'sparse_by_rows' or 'diagonal' has been violated.
in	*n*	is a scalar variable of type int, that holds the number of variables, i.e., the number of rows and columns of S. n must be positive.
in	*S_type*	is a one-dimensional array of type char that specifies the symmetric storage scheme used for the matrix S. It should be one of 'coordinate', 'sparse_by_rows', 'dense' or 'diagonal'; lower or upper case variants are allowed.
in	*S_ne*	is a scalar variable of type int, that holds the number of entries in the lower triangular part of S in the sparse co-ordinate storage scheme. It need not be set for any of the other schemes.
in	*S_row*	is a one-dimensional array of size S_ne and type int, that holds the row indices of the lower triangular part of S in the sparse co-ordinate storage scheme. It need not be set for any of the other three schemes, and in this case can be NULL.
in	*S_col*	is a one-dimensional array of size S_ne and type int, that holds the column indices of the lower triangular part of S in either the sparse co-ordinate, or the sparse row-wise storage scheme. It need not be set when the dense, diagonal or (scaled) identity storage schemes are used, and in this case can be NULL.
in	*S_ptr*	is a one-dimensional array of size n+1 and type int, that holds the starting position of each row of the lower triangular part of S, as well as the total number of entries, in the sparse row-wise storage scheme. It need not be set when the other schemes are used, and in this case can be NULL.

Examples

llsrt.c, and llsrtf.c.

3.1.2.5 llsr_reset_control()

```
void llsr_reset_control (
            struct llsr_control_type * control,
            void ** data,
            int * status )
```

Reset control parameters after import if required.

Parameters

in	*control*	is a struct whose members provide control paramters for the remaining prcedures (see llsr_control_type)
in,out	*data*	holds private internal data
in,out	*status*	is a scalar variable of type int, that gives the exit status from the package. Possible values are: • 1. The import was succesful, and the package is ready for the solve phase

3.1.2.6 llsr_solve_problem()

```
void llsr_solve_problem (
            void ** data,
            int * status,
            int m,
            int n,
            const real_wp_ power,
            const real_wp_ weight,
            int A_ne,
            const real_wp_ A_val[ ],
            const real_wp_ b[ ],
            real_wp_ x[ ],
            int S_ne,
            const real_wp_ S_val[ ] )
```

Solve the regularization-region problem.

Parameters

in,out	*data*	holds private internal data

Parameters

`in,out`	*status*	is a scalar variable of type int, that gives the entry and exit status from the package. Possible exit are: • 0. The run was succesful. • -1. An allocation error occurred. A message indicating the offending array is written on unit control.error, and the returned allocation status and a string containing the name of the offending array are held in inform.alloc_status and inform.bad_alloc respectively. • -2. A deallocation error occurred. A message indicating the offending array is written on unit control.error and the returned allocation status and a string containing the name of the offending array are held in inform.alloc_status and inform.bad_alloc respectively. • -3. The restrictions n $>$ 0 and m $>$ 0 or requirement that A_type or A_type contains its relevant string 'dense', 'coordinate', 'sparse_by_rows' or 'diagonal' has been violated. • -9. The analysis phase of the factorization failed; the return status from the factorization package is given in the component inform.factor_status • -10. The factorization failed; the return status from the factorization package is given in the component inform.factor_status. • -11. The solution of a set of linear equations using factors from the factorization package failed; the return status from the factorization package is given in the component inform.factor_status. • -15. The matrix S does not appear to be strictly diagonally dominant. • -16. The problem is so ill-conditioned that further progress is impossible. • -17. The step is too small to make further impact.
`in`	*m*	is a scalar variable of type int, that holds the number of residuals
`in`	*n*	is a scalar variable of type int, that holds the number of variables
`in`	*power*	is a scalar of type double, that holds the regularization power, p, used. power must be greater than or equal to two.
`in`	*weight*	is a scalar of type double, that holds the regularization power, σ, used. weight must be strictly positive
`in`	*A_ne*	is a scalar variable of type int, that holds the number of entries in the observation matrix A.
`in`	*A_val*	is a one-dimensional array of size A_ne and type double, that holds the values of the entries of the observation matrix A in any of the available storage schemes.
`in`	*b*	is a one-dimensional array of size m and type double, that holds the values b of observations. The i-th component of b, i = 0, ... , m-1, contains b_i.
`out`	*x*	is a one-dimensional array of size n and type double, that holds the values x of the optimization variables. The j-th component of x, j = 0, ... , n-1, contains x_j.
`in`	*S_ne*	is a scalar variable of type int, that holds the number of entries in the scaling matrix S if it not the identity matrix.
`in`	*S_val*	is a one-dimensional array of size S_ne and type double, that holds the values of the entries of the scaling matrix S in any of the available storage schemes. If S_val is NULL, S will be taken to be the identity matrix.

Examples

llsrt.c, and llsrtf.c.

3.1.2.7 llsr_information()

```
void llsr_information (
            void ** data,
            struct llsr_inform_type * inform,
            int * status )
```

Provides output information

Parameters

in,out	*data*	holds private internal data
out	*inform*	is a struct containing output information (see llsr_inform_type)
out	*status*	is a scalar variable of type int, that gives the exit status from the package. Possible values are (currently): • 0. The values were recorded succesfully

Examples

llsrt.c, and llsrtf.c.

3.1.2.8 llsr_terminate()

```
void llsr_terminate (
            void ** data,
            struct llsr_control_type * control,
            struct llsr_inform_type * inform )
```

Deallocate all internal private storage

Parameters

in,out	*data*	holds private internal data
out	*control*	is a struct containing control information (see llsr_control_type)
out	*inform*	is a struct containing output information (see llsr_inform_type)

Examples

llsrt.c, and llsrtf.c.

Chapter 4

Example Documentation

4.1 llsrt.c

This is an example of how to use the package.
Notice that C-style indexing is used, and that this is flaggeed by setting `control.f_indexing` to `false`.

```
/* llsrtf.c */
/* Full test for the LLSR C interface using C sparse matrix indexing */
#include <stdio.h>
#include <math.h>
#include <string.h>
#include "galahad_precision.h"
#include "galahad_cfunctions.h"
#include "galahad_llsr.h"
int main(void) {
    // Derived types
    void *data;
    struct llsr_control_type control;
    struct llsr_inform_type inform;
    int i, l;
    // Set problem data
    // set dimensions
    int m = 100;
    int n = 2*m+1;
    // A = ( I : Diag(1:n) : e )
    int A_ne = 3*m;
    int A_row[A_ne];
    int A_col[A_ne];
    int A_ptr[m+1];
    real_wp_ A_val[A_ne];
    // store A in sparse formats
    l=0;
    for( i=0; i < m; i++){
     A_ptr[i] = l;
     A_row[l] = i;
     A_col[l] = i;
     A_val[l] = 1.0;
     l++;
     A_row[l] = i;
     A_col[l] = m+i;
     A_val[l] = i+1;
     l++;
     A_row[l] = i;
     A_col[l] = n-1;
     A_val[l] = 1.0;
     l++;
    }
    A_ptr[m] = l;
    // store A in dense format
    int A_dense_ne = m * n;
    real_wp_ A_dense_val[A_dense_ne];
    for( i=0; i < A_dense_ne; i++) A_dense_val[i] = 0.0;
    l=-1;
    for( i=1; i <= m; i++){
     A_dense_val[l+i] = 1.0;
     A_dense_val[l+m+i] = i;
```

```
    A_dense_val[l+n] = 1.0;
    l=l+n;
   }
   // S = diag(1:n)**2
   int S_ne = n;
   int S_row[S_ne];
   int S_col[S_ne];
   int S_ptr[n+1];
   real_wp_ S_val[S_ne];
   // store S in sparse formats
   for( i=0; i < n; i++){
    S_row[i] = i;
    S_col[i] = i;
    S_ptr[i] = i;
    S_val[i] = (i+1)*(i+1);
   }
   S_ptr[n] = n;
   // store S in dense format
   int S_dense_ne = n*(n+1)/2;
   real_wp_ S_dense_val[S_dense_ne];
   for( i=0; i < S_dense_ne; i++) S_dense_val[i] = 0.0;
   l=-1;
   for( i=1; i <= n; i++){
     S_dense_val[l+i] = i*i;
     l=l+i;
   }
  // b is a vector of ones
   real_wp_ b[m]; // observations
   for( i=0; i < m; i++){
     b[i] = 1.0;
   }
  // cubic regularization, weight is one
   real_wp_ power = 3.0;
   real_wp_ weight = 1.0;
  // Set output storage
   real_wp_ x[n]; // solution
   char st;
   int status;
   printf(" C sparse matrix indexing\n\n");
   printf(" basic tests of problem storage formats\n\n");
   // loop over storage formats
   for( int d=1; d<=4; d++){
       // Initialize LLSR
       llsr_initialize( &data, &control, &status );
       strcpy(control.definite_linear_solver, "potr ") ;
       strcpy(control.sbls_control.symmetric_linear_solver, "sytr ") ;
       strcpy(control.sbls_control.definite_linear_solver, "potr ") ;
       // control.print_level = 1;
       // Set user-defined control options
       control.f_indexing = false; // C sparse matrix indexing
       // use s or not (1 or 0)
       for( int use_s=0; use_s<=1; use_s++){
          switch(d){
              case 1: // sparse co-ordinate storage
                  st = 'C';
                  llsr_import( &control, &data, &status, m, n,
                              "coordinate", A_ne, A_row, A_col, NULL );
                  if(use_s == 0){
                     llsr_solve_problem( &data, &status, m, n, power, weight,
                                         A_ne, A_val, b, x, 0, NULL );
                  }else{
                     llsr_import_scaling( &control, &data, &status, n,
                                          "coordinate", S_ne, S_row,
                                          S_col, NULL );
                     llsr_solve_problem( &data, &status, m, n, power, weight,
                                         A_ne, A_val, b, x, S_ne, S_val );
                  }
                  break;
              case 2: // sparse by rows
                  st = 'R';
                  llsr_import( &control, &data, &status, m, n,
                               "sparse_by_rows", A_ne, NULL, A_col, A_ptr );
                  if(use_s == 0){
                     llsr_solve_problem( &data, &status, m, n, power, weight,
                                         A_ne, A_val, b, x, 0, NULL );
                  }else{
                     llsr_import_scaling( &control, &data, &status, n,
                                          "sparse_by_rows", S_ne, NULL,
                                          S_col, S_ptr );
                     llsr_solve_problem( &data, &status, m, n, power, weight,
                                         A_ne, A_val, b, x, S_ne, S_val );
                  }
                  break;
              case 3: // dense
                  st = 'D';
                  llsr_import( &control, &data, &status, m, n,
                               "dense", A_dense_ne, NULL, NULL, NULL );
```

```
                if(use_s == 0){
                   llsr_solve_problem( &data, &status, m, n, power, weight,
                                       A_dense_ne, A_dense_val, b, x,
                                       0, NULL );
                }else{
                   llsr_import_scaling( &control, &data, &status, n,
                                        "dense", S_dense_ne,
                                        NULL, NULL, NULL );
                   llsr_solve_problem( &data, &status, m, n, power, weight,
                                       A_dense_ne, A_dense_val, b, x,
                                       S_dense_ne, S_dense_val );
                }
                break;
            case 4: // diagonal
                st = 'I';
                llsr_import( &control, &data, &status, m, n,
                             "coordinate", A_ne, A_row, A_col, NULL );
                if(use_s == 0){
                   llsr_solve_problem( &data, &status, m, n, power, weight,
                                       A_ne, A_val, b, x, 0, NULL );
                }else{
                   llsr_import_scaling( &control, &data, &status, n,
                                        "diagonal", S_ne, NULL, NULL, NULL );
                   llsr_solve_problem( &data, &status, m, n, power, weight,
                                       A_ne, A_val, b, x, S_ne, S_val );
                }
                break;
            }
        llsr_information( &data, &inform, &status );
        if(inform.status == 0){
            printf("storage type %c%1i:  status = %1i, ||r|| = %5.2f\n",
                   st, use_s, inform.status, inform.r_norm );
        }else{
            printf("storage type %c%1i: LLSR_solve exit status = %1i\n",
                   st, use_s, inform.status);
        }
     }
     //printf("x: ");
     //for( int i = 0; i < n; i++) printf("%f ", x[i]);
     //printf("\n");
     // Delete internal workspace
     llsr_terminate( &data, &control, &inform );
    }
}
```

4.2 llsrtf.c

This is the same example, but now fortran-style indexing is used.

```
/* llsrtf.c */
/* Full test for the LLSR C interface using Fortran sparse matrix indexing */
#include <stdio.h>
#include <math.h>
#include <string.h>
#include "galahad_precision.h"
#include "galahad_cfunctions.h"
#include "galahad_llsr.h"
int main(void) {
    // Derived types
    void *data;
    struct llsr_control_type control;
    struct llsr_inform_type inform;
    int i, l;
    // Set problem data
    // set dimensions
    int m = 100;
    int n = 2*m+1;
    // A = ( I : Diag(1:n) : e )
    int A_ne = 3*m;
    int A_row[A_ne];
    int A_col[A_ne];
    int A_ptr[m+1];
    real_wp_ A_val[A_ne];
    // store A in sparse formats
    l=0;
    for( i=1; i <= m; i++){
     A_ptr[i-1] = l+1;
     A_row[l] = i;
     A_col[l] = i;
     A_val[l] = 1.0;
     l++;
```

```
      A_row[l] = i;
      A_col[l] = m+i;
      A_val[l] = i;
      l++;
      A_row[l] = i;
      A_col[l] = n;
      A_val[l] = 1.0;
      l++;
    }
    A_ptr[m] = l+1;
    // store A in dense format
    int A_dense_ne = m * n;
    real_wp_ A_dense_val[A_dense_ne];
    for( i=0; i < A_dense_ne; i++) A_dense_val[i] = 0.0;
    l=-1;
    for( i=1; i <= m; i++){
      A_dense_val[l+i] = 1.0;
      A_dense_val[l+m+i] = i;
      A_dense_val[l+n] = 1.0;
      l=l+n;
    }
    // S = diag(1:n)**2
    int S_ne = n;
    int S_row[S_ne];
    int S_col[S_ne];
    int S_ptr[n+1];
    real_wp_ S_val[S_ne];
    // store S in sparse formats
    for( i=0; i < n; i++){
      S_row[i] = i+1;
      S_col[i] = i+1;
      S_ptr[i] = i+1;
      S_val[i] = (i+1)*(i+1);
    }
    S_ptr[n] = n+1;
    // store S in dense format
    int S_dense_ne = n*(n+1)/2;
    real_wp_ S_dense_val[S_dense_ne];
    for( i=0; i < S_dense_ne; i++) S_dense_val[i] = 0.0;
    l=-1;
    for( i=1; i <= n; i++){
       S_dense_val[l+i] = i*i;
       l=l+i;
    }
   // b is a vector of ones
    real_wp_ b[m]; // observations
    for( i=0; i < m; i++){
       b[i] = 1.0;
    }
   // cubic regularization, weight is one
    real_wp_ power = 3.0;
    real_wp_ weight = 1.0;
    // Set output storage
    real_wp_ x[n]; // solution
    char st;
    int status;
    printf(" Fortran sparse matrix indexing\n\n");
    printf(" basic tests of problem storage formats\n\n");
    // loop over storage formats
    for( int d=1; d<=4; d++){
        // Initialize LLSR
        llsr_initialize( &data, &control, &status );
        strcpy(control.definite_linear_solver, "potr ") ;
        strcpy(control.sbls_control.symmetric_linear_solver, "sytr ") ;
        strcpy(control.sbls_control.definite_linear_solver, "potr ") ;
        // control.print_level = 1;
        // Set user-defined control options
        control.f_indexing = true; // Fortran sparse matrix indexing
        // use s or not (1 or 0)
        for( int use_s=0; use_s<=1; use_s++){
           switch(d){
               case 1: // sparse co-ordinate storage
                   st = 'C';
                   llsr_import( &control, &data, &status, m, n,
                               "coordinate", A_ne, A_row, A_col, NULL );
                   if(use_s == 0){
                      llsr_solve_problem( &data, &status, m, n, power, weight,
                                          A_ne, A_val, b, x, 0, NULL );
                   }else{
                      llsr_import_scaling( &control, &data, &status, n,
                                           "coordinate", S_ne, S_row,
                                           S_col, NULL );
                      llsr_solve_problem( &data, &status, m, n, power, weight,
                                          A_ne, A_val, b, x, S_ne, S_val );
                   }
                   break;
               case 2: // sparse by rows
```

```
                st = 'R';
                llsr_import( &control, &data, &status, m, n,
                             "sparse_by_rows", A_ne, NULL, A_col, A_ptr );
                if(use_s == 0){
                   llsr_solve_problem( &data, &status, m, n, power, weight,
                                       A_ne, A_val, b, x, 0, NULL );
                }else{
                   llsr_import_scaling( &control, &data, &status, n,
                                        "sparse_by_rows", S_ne, NULL,
                                        S_col, S_ptr );
                   llsr_solve_problem( &data, &status, m, n, power, weight,
                                       A_ne, A_val, b, x, S_ne, S_val );
                }
                break;
            case 3: // dense
                st = 'D';
                llsr_import( &control, &data, &status, m, n,
                             "dense", A_dense_ne, NULL, NULL, NULL );
                if(use_s == 0){
                   llsr_solve_problem( &data, &status, m, n, power, weight,
                                       A_dense_ne, A_dense_val, b, x,
                                       0, NULL );
                }else{
                   llsr_import_scaling( &control, &data, &status, n,
                                        "dense", S_dense_ne,
                                        NULL, NULL, NULL );
                   llsr_solve_problem( &data, &status, m, n, power, weight,
                                       A_dense_ne, A_dense_val, b, x,
                                       S_dense_ne, S_dense_val );
                }
                break;
            case 4: // diagonal
                st = 'I';
                llsr_import( &control, &data, &status, m, n,
                             "coordinate", A_ne, A_row, A_col, NULL );
                if(use_s == 0){
                   llsr_solve_problem( &data, &status, m, n, power, weight,
                                       A_ne, A_val, b, x, 0, NULL );
                }else{
                   llsr_import_scaling( &control, &data, &status, n,
                                        "diagonal", S_ne, NULL, NULL, NULL );
                   llsr_solve_problem( &data, &status, m, n, power, weight,
                                       A_ne, A_val, b, x, S_ne, S_val );
                }
                break;
            }
        llsr_information( &data, &inform, &status );
        if(inform.status == 0){
            printf("storage type %c%1i:  status = %1i, ||r|| = %5.2f\n",
                   st, use_s, inform.status, inform.r_norm );
        }else{
            printf("storage type %c%1i: LLSR_solve exit status = %1i\n",
                   st, use_s, inform.status);
        }
     }
     //printf("x: ");
     //for( int i = 0; i < n; i++) printf("%f ", x[i]);
     //printf("\n");
     // Delete internal workspace
     llsr_terminate( &data, &control, &inform );
   }
}
```